THE BELL SYSTEM TECHNICAL JOURNAL
Vol. 56, No. 7, September 1977
Printed in U.S.A.

No. 4 ESS:

Program Administration, Test, and Evaluation

By P. S. McCABE, J. B. OTTO, S. ROY, G. A. SELLERS, JR., and K. W. ZWEIFEL

(Manuscript received July 8, 1976)

A number of hardware and software support systems are used in the design, testing, evaluation, and administration of the No. 4 ESS software. The Program Administration System (PAS) provides a management facility for the large data base that consists of the multitude of ESS programs. These programs must be changed, controlled, and manipulated in a uniform fashion to produce the various official releases of the No. 4 ESS software without jeopardizing the integrity of the data base. The 1A Processor Utility System uses a minicomputer and a hardware interface with the No. 4 ESS as an operating system for the interpretation, initialization, execution, monitoring, and display of tests written in a high-level utility language. Thus it makes available a convenient means for testing and debugging No. 4 ESS software in a system laboratory environment. A directed-graph model, describing the sequential stimulus-action-transition behavior of a software process under test, acts as a reference and a starting point for the Automated Testing and Load Analysis System (ATLAS). Components of ATLAS then automatically derive tests and employ test directives embedded in the model to apply and monitor tests for the modeled process. Testing of the ESS software under high traffic loads is achieved in the laboratory by the Programmed Electronic Traffic Simulator (PETS). This system simulates the peripheral equipment and monitors the ESS responses for deviations from a norm.

I. INTRODUCTION

A large development undertaking like No. 4 ESS—one that spans years, involves several organizations, and is built by the combined efforts of many people—is seldom described solely in terms of the final mar-

keted product. Numerous supporting activities must exist to assist the project through its design, implementation, testing, evaluation, and production of official releases. A number of support tools that were developed to meet the demands of producing quality software in No. 4 ESS in a systematic and efficient fashion evolved to be systems of significant sizes themselves. A common objective behind these support systems was to provide their users, viz., the community of No. 4 ESS programmers, with powerful yet easy-to-use tools to assist in the development of a quality product.

Four such systems of diverse scope, content, and applicability but linked by the common objective above are described herein. The Program Administration System (PAS) provides management facilities for the sources, object modules, listings, and other data entities associated with the many programs constituting the No. 4 ESS software package. PAS is described in Section II. In Section III, the 1A Processor Utility System that enables program testing and debugging to be done in the controlled environment of the System Laboratories is described. A novel testing concept and its application methodology and component tools, collectively known as the Automated Testing and Load Analysis System (ATLAS), is described in Section IV. While ATLAS primarily concentrates on testing the complex logic of the programs and their interfaces, engineering aspects of the software and dynamic performance are evaluated in the face of various levels of system loading by the Programmed Electronic Traffic Simulator (PETS). PETS is described in Section V. Each section is essentially complete in itself. The objectives, components, features and usage experience related to a system are covered in its associated section.

II. PROGRAM ADMINISTRATION SYSTEM

2.1 General

It was anticipated that the object code of the No. 4 ESS software system might reach 1,000,000 words in size, that the number of separately assembled programs (PIDENTs) would be 1000 or more, that over 200 program designers would be involved, and that the volume of textual documentation associated with the software would exceed 100,000 pages. It was clear that no system of manual procedures, no matter how vigorously enforced, would suffice to keep order and accountability in the creation and administration of the No. 4 ESS software. An all-inclusive system of facilities for both programmers and administrators was needed.

Further, the No. 4 ESS software would continue to evolve. New features and program changes would be incorporated into new generic programs (GENERICs) while older GENERICs would simultaneously be supported.

The No. 4 ESS software system consists of two major components: the No. 4 ESS application software and the 1A Processor software. The 1A Processor software is used in both the No. 4 ESS and the No. 1A ESS (a local switching system utilizing the 1A Processor). These two pieces of software were developed concurrently but in two different organizations. Hence there was a need to administer them as two separate but closely coordinated communities. A further administrative division came from the need to provide special facilities for the diagnostic maintenance programs, which were used in the generic programs, and for the Installation Test System used by Western Electric Company engineers in the installation of No. 4 ESS offices. Finally, the support computing system, the IBM 360/Time Sharing System known as TSS, was anticipated to have insufficient capacity to accommodate the entire development in a single system. Hence the administration of the software development was to be carried out in several separate TSS systems. Within a single support computation system, it would be useful to allow several communities of users to be recognized and to be able to divide the data base along administrative boundaries. The Program Administration System (PAS) was designed to address the problems of administering the large software system of the No. 4 ESS.

2.2 Objectives

The primary objective of the Program Administration System is to provide for the creation, modification, preservation, and administration of the No. 4 ESS and 1A Processor software and their associated data bases. This implies an orderly structure for all data (both temporary and permanent) and a set of facilities which enable the creation of data, the copying and erasing of data, the naming and renaming of data sets, and the combining of data sets to form new data sets. These facilities must allow for moving data between various types of storage and must include reporting which is necessary to keep the users up to date on the state of their data.

The second objective is to allow the user and the administrator* easy and efficient access to the data base and to the support programs which are used in the development and testing of each PIDENT. Access must be concise to eliminate ambiguity over what is being accessed. Access must also be simple to minimize errors in the user's specification. In the case of data sets, a structured and functional naming convention is needed. This allows the user to readily determine the name of each data set through knowledge of the function of the data set. In the case of support programs, the calling sequences and parameters must be uni-

* The administrator is responsible for the production of the GENERIC from the multitude of component parts known as PIDENTS.

form so the user can easily remember them. There is also a need to provide values (termed "defaults") for parameters when the user does not supply values. The use of defaults is a powerful administrative tool which allows the system to produce correct calling sequences even when the user does not know the correct values.

The third objective is to provide the administrator with the facility to manipulate the data base on a large scale to accomplish backup protection against both system failure and data mutilation due to inadvertent destructive actions on the part of users or administrators. It is important that these facilities be comprehensive and foolproof.

The fourth objective is to make maximum use of the basic TSS facilities in the implementation of the Program Administrative System. This approach minimizes the need to create new basic facilities and assures a robust design in the face of changes in TSS. An additional benefit is the consistency in appearance this produces between the Program Administration System and its host, TSS.

The fifth objective is to make efficient use of computing resources. Since the Program Administration System has a large number of users and has a heavy influence on how they use the computing facilities, it is important that its usage be efficient. In a similar vein, the administrative environment should discourage duplication of storage (for example, users making their own backups to protect their data) and processing.

The final objective is to implement the system in a modular and extensible structure. The Program Administration System functions within an evolving environment and frequently must accommodate change so that users are not impacted. It is important that the system be designed to be readily modified and extended to handle changes.

2.3 User facilities

The user facilities provided by the Program Administration System fall into three categories: facilities which support the basic conversational and batch processes that are associated with PIDENT development, facilities which allow the user to create and save or execute batch jobs while engaged in conversational processing, and facilities which provide administrative and general-purpose computing capabilities.

The system provides the facility to edit PIDENT source data sets, cause assembly of a PIDENT (inputting the PIDENT source data set to the assembler), load the Output Program Module (OPM), simulate the execution of the loaded PIDENT OPM, or execute any meaningful combination of those processes by issuing a single terse command supplied with a single parameter: the PIDENT name. The proper data sets will be accessed, modified, or created in the Program Administration System data

base by each process without further attention from the user. For the assembly process, No. 4 ESS and 1A Processor software systems utilize a set of libraries called COMPOOLS which contain data definitions, data structures, and macro definitions. Each PIDENT utilizing any of these named values or structures obtains the correct definitions at assembly time via access to the correct COMPOOL. The Program Administration System assures that the proper COMPOOL is made available to each assembly.

A further extension supported by the Program Administration System is the concept of a PROGRAM UNIT, or a part of a PIDENT which may be assembled separately. The system allows the user to create an arbitrary collection of PROGRAM units, and to operate on each of them as though they were PIDENTS. The user can combine any group of PROGRAM UNITS into a single PROGRAM UNIT or into a PIDENT at will. This facility allows the user to develop a PIDENT by building it from smaller functionally oriented pieces.

A full set of batch facilities, all of which can be invoked while the user is in the conversational mode, are provided. The user can build data sets which consist of batch jobs (which may include both PAS and non-PAS commands and processes) and can execute these jobs concurrently with the conversational jobs. The user can create a series of such jobs and be guaranteed of the sequence of execution. The user will be informed of the batch job beginning and completion via messages sent from the batch job to the user's conversational job. If the user attempts to exit from the Program Administration System without specifying the disposition of any batch job created, he will be prompted for the proper disposition (execute, save, or erase).

The user is provided with access facilities other than the standard processes previously described. For instance, the user can create "private" versions of PIDENTs or PROGRAM UNITs and can maintain them separately from the official versions. The Program Administration System provides a facility which indicates the elements to be included in the official version. This facility is accessed by the administrator when assemblies are done for system loads.

A user may issue TSS commands while using the Program Administration System and can temporarily leave and return without invoking the overhead that is associated with a user's initial access. No TSS facilities are denied the user under the Program Administration System except those which actually conflict. More details on these issues will be given in Section 2.5.

2.4 Administrator facilities

The administrator facilities are primarily directed to the task of organizing and processing the PIDENT-associated data in accordance with

the needs of the project as a whole. In a more subtle vein, the administrator facilities must provide the ability to redirect or change the results of user processes and to change the details of the facilities available to the users. Finally, the administrator must have tools to create and permute the administrative data itself and to use that data to control user processes and administrator processes.

The Program Administration System provides a data base which is owned by the administrator and made available to the user on a shared basis. The administrator can permit access on a read-only basis, a read/write basis, or an unlimited basis (the user can create or destroy the data set); or the administrator can deny all access. The permission can be established at a data set level or over groups of data sets. The system supports many administrative communities, allowing each administrator autonomous control over a community, but allowing the transfer of data between administrative communities at the administrator level. Hence the No. 4 ESS administrator can gain access to the 1A Processor community in order to prepare a generic load for No. 4 ESS.

The administrator has the facilities to build the structure of the data and processes necessary to add a new user to the Program Administration System and to build the administrative data which would allow the new user to access all necessary PIDENT data. The administrator can reestablish all of these structures from data saved in the administrative data base if some system disaster should strike. The entire data base (including the administrative data) is protected by a powerful data base backup facility executed by the administrator. This facility allows the administrator to select the criteria for data set backup and the frequency of backup. The data base resides on a different media than the backup, which makes it unlikely that the backup can be damaged by the same mechanism which damages the data base.

The administrator has the same facilities as any user. Hence the administrator can perform tasks on behalf of the user, or even in spite of the user when necessary. In addition, the administrator has free access to all PIDENT data and can execute processes which are denied to any user. This is a heavily used facility when the software under development is under strict change control (termed "frozen"). Under these circumstances, only the administrator can modify a PIDENT, and users may make modifications only through the change control process.

The administrator also has the facility to modify the way in which a user accesses the support programs and to determine which version of a support program is made available to users. This facility further allows the administrator to introduce changes into the user environment while the user is using the Program Administration System.

In addition to the facilities for controlling user access to the data base,

which are described earlier in this section, the administrator has the overall capability to control the allowable syntax and semantics in that community and to control exactly which facilities are invoked with each user or administrator command. The definition of most processes and the parameters passed to each process (including default values) can be modified by the administrator.

When changes are introduced that require user notification, the administrator can issue messages to all conversational users and can send messages to be included in all batch jobs. An announcement can be sent to each user as they log into the system. There is a "news" facility which the user can use to obtain general information supplied by the administrator.

For changes that require that the system not be in use, the administrator can invoke controls that prevent further conversational and batch jobs from beginning. The administrator can also query to determine the identity of any users logged in. The administrator can then send messages to the conversational users requesting them to leave the system (or even issue the command which logs the user off), and can wait until the executing batch jobs finish before introducing the changes into the quiescent system.

2.5 Implementation

The Program Administration System is implemented on and executes under the IBM 360/67 Time Sharing System, which is a large general-purpose conversational computing system. The principle characteristics of this operating system are:

(*i*) A very large addressing space (16 million bytes) available to each user, called "virtual" memory.

(*ii*) A time-shared operating system which gives seemingly continuous service to reasonable numbers of concurrent conversational users.

(*iii*) A powerful and flexible command system which includes the capability to build, preserve, or delete individual user commands.

(*iv*) A "default" facility, which allows prespecified values to be supplied to parameters of commands when the command has not supplied values explicitly.

(*v*) A heirarchical catalog which allows each user to create, access, and erase data sets by name, and allows users to share each other's data sets. The sharing permission is granted by the owner and can be read-only, read/write, or unlimited.

(*vi*) A group of data-set access methods which deal with the complexities of concurrent access of a data set by several users and control the interwrite problem using a system of access locks.

(*vii*) A complete set of batch-processing facilities and a capability which allows users to initiate batch jobs while in the conversational mode.

(*viii*) A dynamic loader which automatically loads programs invoked by any command. The TSS and the user provide libraries to the loader which are searched for programs in response to commands or to resolve references in programs being loaded.

The user of TSS is supplied with an environment of commands, defaults, and facilities (programs) when initially joined to the system. The user may subsequently add to that environment or change the values of the defaults or the function of the commands and may elect to save these changes for future use or let them disappear at the conclusion of the current session. An address space and identity known to TSS as a task is created for each user when he logs onto TSS either conversationally or as a batch job. At a given time, a user may have in existence one conversational task plus as many batch tasks as he has concurrently executing. When the user logs off TSS or his batch job completes, the associated task disappears. Changes to the user's environment during the execution of the task are normally lost when the task ends, but can be preserved by explicit command of the user.

The Program Administration System operates as a subsystem under TSS, and a major element in the design was the attempt to keep a consistent appearance to the user. In many cases, TSS did not have facilities to deal with the concept of a subsystem and much of the significant design work involved the creation of these facilities (most of which were then incorporated into TSS for the use of other subsystems).

When a user first accesses the Program Administration System (a process called "joining"), an environment is built for him. In subsequent accesses, the user will have this environment merged into the TSS environment, and any conflicts (commands or defaults with identical names) will be resolved in favor of the Program Administration System environment. The Program Administration System will interpret and execute all commands issued by the user. The user may execute any TSS command by prefixing the command with an underscore, which will cause the system to pass the command to TSS for interpretation and execution. Similarly, the user may execute a command of his own creation by prefixing the command with a dollar sign, which will cause the Program Administration System to restore the user's original environment and then pass the command to TSS for interpretation and execution. The user may also exit from the Program Administration System on a temporary basis to do work in TSS and then return without causing the reestablishment of environments.

After command interpretation by the Program Administration Sys-

tem, the result is usually a call to invoke a program and a stream of parameter values to be passed to that program. As described earlier in this section, the TSS dynamic loader resolves the program call by searching a list of libraries (called the JOBLIB chain) which has been established by TSS and the user. When the program is first found (the program may appear more than once in the chain of libraries), it is loaded and its parameters are passed to it before execution. If there are parameters expected by the program for which no values have been passed, TSS uses the default from the task environment for the value (if one exists). The Program Administration System utilizes these facilities by supplying a library or series of libraries to be put at the *top* of the JOBLIB chain for the user's task and by supplying a set of default values to be merged into the task environment. The administrator can change the libraries and default values freely, and the user is given access to a portion of the defaults to allow the tailoring of his environment. Further, the Program Administration System rarely makes a direct call to a support program; instead it issues a command. These commands are built through the TSS command-building facility and are part of the environment which is merged into the user's task. These commands can be easily altered by the administrator and provide power and flexibility in the mechanism which invokes support programs. Since these commands are used to pass parameter values to the support programs, it is possible to override the parameter values that the user has supplied or those that are default values.

Some programs expect to receive their input from the user's terminal or from a data set which will be identified in the terminal input. TSS uses a mechanism called the "source list" to serve as a buffer between the terminal data and the program which receives it. The Program Administration System manipulates the source list when it is necessary to provide input to such a program before the user begins his input or when it is necessary for the Program Administration System to "get in the last word."

The conversational user may use the "attention" key on his terminal to gain control over program execution, that is, to interrupt the execution of a program. When the interruption is accomplished, TSS will allow the user to carry out some other activity and then resume (if the user desires) the interrupted execution. TSS provides the facility to allow the interrupted program to recognize the attention, and provide its own response (called "fielding the interrupt"). The Program Administration System utilizes this facility extensively to allow or deny each support program the fielding of the interrupts during its execution. Further, the system may also field other types of interrupts such as those caused by input/output or system routines. The objective of this approach is to provide the user with a reasonable response to an attention at any phase of

process execution, and to shield the user from confusing system responses. Finally, the Program Administration System makes extensive use of the TSS interrupt-handling facilities to provide task-to-task communications. These capabilities include administrator announcements to all users, administrator-user messages, user-user messages, notification messages to a conversational user regarding batch job activity, and statistics gathering.

The data base for the Program Administration System is implemented entirely in direct-access disk storage. The majority of this storage is on private disk packs, although most storage in TSS is implemented in a common pool of direct-access disk storage called "public storage." The use of private disk packs provides the freedom to administer the storage independently between communities and independently of the rest of the TSS users. It also allows the user the option to keep his general storage separate from the project data base, and allows a backup-and-restore facility which can adopt administrative boundaries. Backup of this data base is done to magnetic tape on a regular basis by the administrator, and restoral of a single data set or the entire data base can be done by the administrator. Backup is currently done daily on all data sets changed that day, and the entire data base is backed up weekly. All project-oriented data generated under the Program Administration System and some project-oriented data generated within our support systems is maintained in this data base and protected through the backup procedures.

The Program Administration System command analysis is performed in a program which executes the lexical and syntactic analysis from a data table produced by a compiler-compiler. The lexical and syntactic rules were compiled into a data table, and this table is used to drive the analyzer. The result of this analysis is a data structure (a job stack) and a table of values (hash table), which can be used by the interpreter in the execution of the job stack. The interpreter program operates on the job stacks to produce calls to programs or commands in TSS, together with the parameter values to be passed to those processes. The lexical and syntactic rules and the values of constants needed for the hash table are stored in an administrative data set for each administrative community and are used to build the user and the administrator environment when each accesses the Program Administration System.

2.6. Commentary

For several years the Program Administration System has been used continuously by several projects on two separate IBM TSS systems.

The objectives set for it were met, and there is no doubt that chaos would have reigned without such a system. Several observations should be made on the design approaches and the objectives themselves.

The decision to implement the system as a subsystem of TSS was a good one. Much design effort was saved by minimizing the reinvention of many facilities already available in TSS, and in many cases the extensions made for the Program Administration System were useful to other TSS users. TSS evolved through many software releases and through conversion to the IBM 370 series of hardware during the development period, and the impact of this on the Program Administration System was minimal. There was virtually no impact on the Program Administration System users from these TSS changes. The users of TSS and of the Program Administration System benefited from the consistency of appearance and function.

Although much was done to make the structure of the Program Administration System flexible and very general, not enough generality was achieved. It is now clear that all of the implications of generality may still not yet be known. A measure of generality which would allow an administration system to create subsystems within itself, each with all of the facilities of its progenitor, seems to be a good start on this problem. The Program Administration System was able to meet all of its original requirements, and was able to adapt to many new and unexpected ones, but eventually the necessity to use a "kludge" to solve an immediate problem produced a less and less modifiable system.

Finally, it is clear the entire software development process, not just the program-writing phase, is in need of administration. Hence there is a need for a development support system which would provide facilities for all phases of software development.

III. 1A PROCESSOR UTILITY SYSTEM

3.1 General

Convenient and efficient debugging tools were required for the development of the large body of software that would be resident in the 1A Processor. As mentioned earlier, the 1A Processor itself was being developed for application in two major Electronic Switching Systems:

(*i*) The 1A ESS designed to serve customers at the local level.
(*ii*) No. 4 ESS designed to serve the national network as a toll switch.

Because of this dual application, it was necessary to design a utility system which would serve both the common and individual needs of the two systems.

Since the ESS laboratories represented millions of dollars of invested capital, laboratory time was a precious development resource. To meet schedule goals, a total of four system laboratories utilizing the 1A Processor were initially constructed. Two of these were No. 4 ESS labora-

tories and two were 1A ESS laboratories. Because there was neither floor space nor money for a fifth laboratory, efficient utilization of the existing four laboratories was a necessity. Efficiency here means the time spent in test execution relative to the total laboratory time devoted to that test. Total time includes the overhead of setting up the test and of outputting test results.

To enhance user productivity, tools had to be provided that would give the most useful results in the form of run-time data with the least amount of personnel effort and time. These objectives led directly to development of the 1A Processor Utility System which would allow a batch mode of operation wherein the user could leave his test for an operator to execute.

The input language for the utility system user needed to be intuitively straightforward, and symbolic representation had to be permitted to the greatest possible extent. Output data needed to be conveniently formatted, with symbolic representation employed wherever possible.

Since data collected from a real-time multiprogram environment can have a high degree of time dependency, there are circumstances where the act of data collection should not interrupt the operation of the machine under test. Autonomous collection of data from key processor locations was therefore an important feature that needed to be provided by the 1A Processor Utility System.

To meet these needs, the 1A Processor Utility System employs an SEL 810B minicomputer with a vendor-supplied real-time monitor system. It is programmed to provide both user and administrative functions which are described herein. There is a complex hardware interface known as the Utility Test Console (UTC), and finally there is a body of utility code in the 1A Processor itself.

3.2 Hardware configuration

The hardware configuration of the 1A Utility System is explained through use of Fig. 1.

The upper part of the figure shows the main elements of the 1A Processor. The central control is the central processing unit, and it is fully duplicated. There are four bus systems. Generally speaking, the call stores on the call store bus contain time-variant data, whereas the program stores on the program store bus contain invariant data including the program itself. Both of these stores are constructed of magnetic cores. The Auxiliary Unit (AU) bus serves magnetic disk units via the disk file controller. Major peripheral units of ESS, including the man-machine interface known as the Master Control Center, are served by the peripheral unit bus.

The lower part of Fig. 1 shows the major units of the utility system.

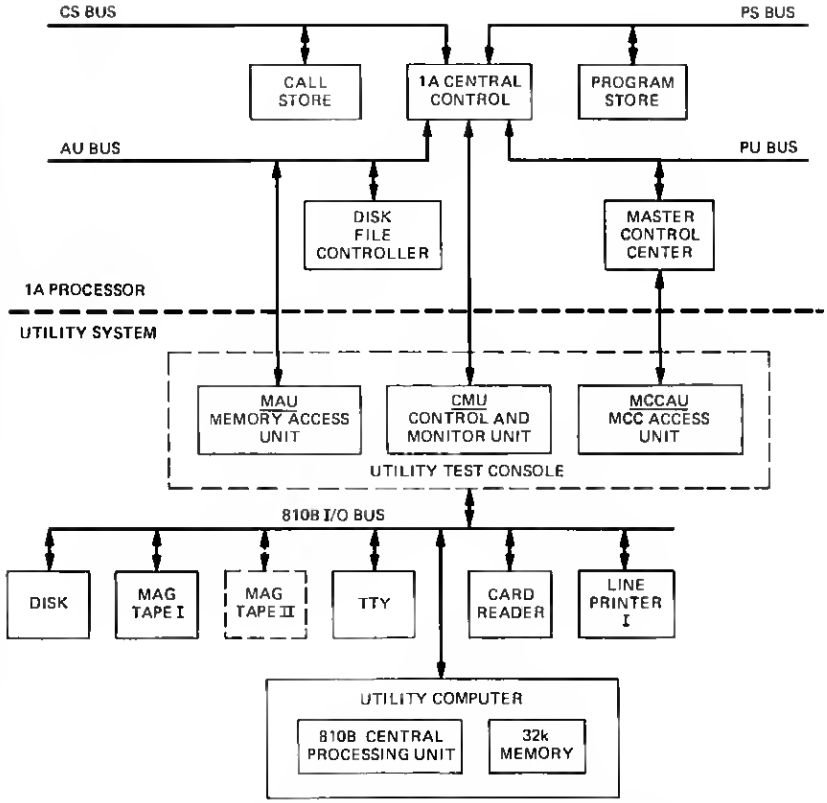

Fig. 1—1A Processor utility system hardware configuration.

The UTC serves as an interface between the 1A Processor and the SEL 810B computer.

The UTC consists of three functionally independent units: the Memory Access Unit (MAU), the Control and Monitor Unit (CMU), and the Maintenance Control Center Access Unit (MCCAU).

It is through the MAU on the auxiliary unit bus of the 1A Processor that the Utility Computer (UC) is able to read and write ESS memory locations.

The CMU contains registers which shadow a large number of central control locations. These registers in turn provide an input to matchers that can trigger a utility action when a predetermined condition exists. There is also a monitor store with capacity for 512 entries resulting from snapshots of the contents of processor locations. A single snapshot entry consists of a total of 528 bits of data. The data gathered on each snap consists of a fixed part and a variable part. The variable part is a set of up to four quantities chosen by the user from a list of 21 possibilities and specified in the job deck at the start of the run.

The MCCAU allows interrogation of processor status indicators and the setting of major controls on the MCC.

The utility computer has access to the subunits of the UTC via its I/O bus, which is time-shared with its own peripheral units. As shown in Fig. 1, the utility system peripherals consist of two disks (one with 2-megabyte capacity and the other with 6-megabyte capacity), a magnetic tape unit with an optional spare, a card reader, a line printer, and a typewriter.

The user input to the utility system is usually via the card reader, and normally the test results are printed on the line printer.

All utility system run-time actions occur as a result of a user-specified trigger. The trigger may either be a hardware trigger in the UTC or a software trigger resulting from planting a branch instruction at a specified program address which will interrupt the normal program flow and transfer control to the 1A Processor resident utility program. Following a hardware trigger, there may be autonomous actions such as a snapshot of processor locations, or, as in the case of the software trigger, there may be embedded actions requiring the execution of utility code in the 1A Processor. An example of an embedded action is the memory dump which moves a user-specified block of memory to space reserved for the utility system in call store. At the conclusion of the test, the contents of the utility call store, together with data in the monitor store in the CMU, are formatted and printed for the user.

As mentioned previously, run-time actions may be caused by a hardware trigger within the UTC. A variety of matchers have been provided which allow great flexibility in the choice of conditions that will initiate utility action.

The hardware matchers are provided in the CMU portion of the UTC. The matcher inputs are on the one hand a user specified quantity and on the other hand an address or data input from the central control.

There are 16 single address matchers. Since the matchers are independent of each other, independent utility actions may be specified for up to 16 individual addresses in a single test run. Twelve matchers may be set to match on either an instruction or data address residing in either the current program store address register or the data address register, respectively, of the central control. The other four may be set to match on a core address used in transfers to and from a unit on the AU bus (generally a file store). When a data address has been specified, the user has the additional option of qualifying the match condition to produce a trigger on a read, a write, or either a read or write operation.

Eight block address matchers have been provided. Six are for matching against instruction or data addresses obtained from the processor using the same register sources as the single address matchers. The other two are used to match against core addresses used in transfers to or from a

unit on the AU bus (generally a file store). As implied by the name, a block address matcher requires inputs to specify both the start and end address of the block. There are two selectable matching modes: the interface mode and the block mode. In the interface mode, a trigger is produced the first time the source address lies within the user-defined block or, optionally, the first time the address lies outside the specified block. A third option produces a trigger when either entering or exiting the address block. In the block mode, the user has the additional option of requesting whether the addresses of interest are inside or outside the block. In this mode, a trigger is generated and associated utility action results each time an address inside (or, optionally, outside) the specified address block is referenced.

Four 24-bit data matchers have been provided. These may be used to match against the contents of any of 16 central control registers. The match may be against all of the bits or against any combination of bits as specified by a masking quantity.

There are two counter matchers, each of which may count one of six user-specified count sources. The user specifies the source (e.g., processor cycles) and the value of the count which will produce a trigger.

An interrupt matcher is provided that produces a trigger each time a selected processor interrupt level occurs. Any combination of levels may be specified.

Finally, there are four multiple-condition matchers which will produce a trigger when a specified combination of any of these conditions occurs. The trigger and associated utility action will occur only when the specified combination occurs in the same instruction cycle.

3.3 Utility language

Through the use of the utility language, the user has the ability to initialize the test environment, to control the execution of programs, and to specify the collection of test data. The statement is the basic component of the language, and there are two basic types:

(*i*) Job control statements, which delimit the job deck and establish the operating environment

(*ii*) Utility language statements, which determine the utility actions that are to occur during the test run.

All statements have a command field, except comment-only statements. In the case of the utility language statement, the command field is further subdivided into a trigger clause and an action clause. The trigger clause specifies the conditions that will initiate utility actions, and the action clause specifies the utility actions that are to occur. Conditional action

clauses based on run-time data are permitted. So that run-time decisions may be made to enable or disable a statement by name, a name label may accompany the utility language statement.

The basic purpose of job control commands is to establish the environment in which the test will be executed. The user may choose to accept the system in the state left by the previous job or may specify with considerable exactness the hardware and software configuration desired. Overwrites may be installed prior to the start of the test, and the user has the option of expressing them either in mnemonic terms or in octal.

Trigger commands specify the conditions that will initiate utility action. As explained previously, the trigger may be either a hardware trigger resulting from a specified condition detected by a hardware matcher or a software trigger resulting from a branch instruction planted at a specified program address. There are subtle but important differences between these two trigger types. The software trigger operates on the machine state existing just prior to the branch instruction and interferes with the program flow. Hardware triggers are completely autonomous in themselves, although the resulting utility action may not be. Autonomous action that may accompany a hardware trigger consists of either snapping key central control locations or initiating or terminating a trace action. The trace terminates either by command or when the trace counter is satisfied. A useful feature is the trace-last option whereby only a user-specified number of the last instructions (or branches) of the trace are output.

The action clause specifies the utility actions that are to follow a trigger. A variety of utility verbs may be specified in the action clause. Generally speaking, any combination of verbs may accompany any trigger. Specified values may be set into registers or memory locations. The contents of registers or memory locations may be saved for output at the end of the job. Other utility statements may be enabled or disabled. Program execution may be transferred to a user-specified address. Conditional action based on the current machine state is also possible. Instruction or transfer traces may be initiated or terminated. Messages may be input to ESS or sent to the system operator. This last action is useful when it is necessary to ask for manual action during the test.

A particularly useful capability available to the user is that of writing new utility functions and then invoking them during the test using any of the available triggers. This feature, known as the DOIT function, has enabled the users to meet unique needs which are not satisfied by the general utility verbs.

Pseudo operations do not directly generate any utility functions, but are a necessary part of the language to improve its convenience. Pseudo operations may be used to define symbols, to change default values, to

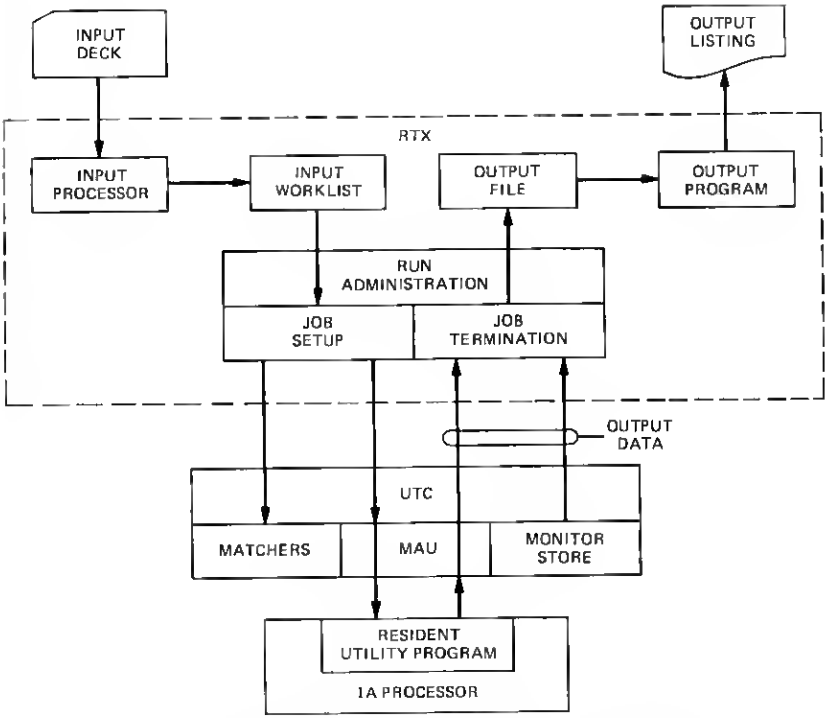

Fig. 2—1A Processor utility system software organization.

specify the start address and range of a set of overwrites, or to specify the use of a prestored set of utility statements.

3.4 Personal mode

Another mode of operation available to users is known as the personal mode. This mode gives the user the benefits of interactive operation; and though the personal mode commands are limited, they permit direct manual control. Typically, a user will input a set of commands for a conventional batch job and then invoke the personal mode. Through use of personal mode commands, the utility system may output any data immediately after it is collected instead of at the end of the test. Optionally, data collected may be erased, specified memory locations may be dumped, register or memory locations may be set to a specified value, or program execution may be transferred to a specified address or the test may be terminated.

3.5 Software organization

The software organization of the 1A Processor Utility System is explained through use of Fig. 2. The Real Time Executive (RTX) is a ven-

dor-supplied software monitor controlling the utility programs. The utility system software consists of four major programs: input processor, run administration, 1A Processor resident utility program, and output program. The input processor interprets the input and generates a worklist. Run administration sets up the job according to commands in the worklist, controls the job during execution, and collects the output data after the job terminates. During execution of the job, the 1A Processor resident utility program performs nonautonomous user-specified utility actions. The output program formats the data, outputs the listing and saves statistical data about the job.

The input processor program reads and interprets the input, generates a worklist corresponding to the input statements, and records the input as part of the information contained in the output file. Due to memory limitations, the input processor program is divided into three tasks which are executed sequentially. The first task reads the input and performs a syntactic scan of all statements. The second task performs a semantic check and builds a worklist for all job control statements. The third task performs a semantic check and builds a worklist for utility language statements. Run administration, the next utility system task, is activated by the input processor program.

Run administration is responsible for controlling the job during setup and execution, and collecting the data after the job terminates. Run Administration sets up the job according to commands stored in the worklist. Typical job setup functions include configuration and initialization of ESS, loading the 1A Processor system from magnetic tape, installing overwrites, initializing the UTC hardware matchers, installing branch instructions in ESS programs to implement the software triggers, and passing tables to the 1A Processor resident utility program for execution-time processing. During job execution, run administration times the job, responds to execution-time requests for activation or deactivation of hardware matchers, responds to job termination requests, and responds to user commands from the operator's console if the job is a personal-mode job. After the job terminates, data generated during the job must be collected. Autonomous traces and snaps are saved in the monitor store of the UTC. Data generated from nonautonomous action is saved in a call store dedicated to the utility system. This data is collected and merged in a time-sequential output file. Lastly, overwrites and software triggers are removed, and the original instructions are restored. Run administration is now completed and the Output Program is activated.

The 1A Processor resident utility program is entered after a trigger to execute any nonautonomous action. The program is primarily table driven, using tables passed by Run Administration. The work table contains encoded data about the specific requests for each trigger. An

entry vector table contains a pointer to the work table which was constructed for each trigger. For each software trigger, the return table contains the original instruction and the return address where ESS program execution will be resumed.

The 1A Processor has programs which are paged into core and overlay a portion of memory. Since utility requests may apply to paged programs, the 1A Processor resident utility program monitors the programs as they are paged and activates or deactivates the necessary triggers.

The output program formats the raw data from the output file, outputs the job listing, and saves statistics accumulated for the job. The addresses of the triggers are converted to a symbolic name plus displacement. Finally, the output program activates the input processor for the next job.

3.6 Administrative and support features

As in any programming system, the versatility of the system depends greatly on support programs. The utility system is no exception and has numerous support programs which may be grouped into five functional types: loader map administration for symbolic name representation, overwrite administration, statistic gathering, message handling, and operator programs.

Program designers prefer to specify a program address as a symbolic program name (base address) plus displacement, rather than as an absolute number. Similarly, faster evaluation of the output is possible if the addresses are printed as a symbolic program name plus displacement. The loader process produces additional files on the loader tape for the benefit of the utility system. These files, called loader map files, are stored on the utility system disk. The input processor program references one file for name-to-address conversion. This file contains all program and entry names. Another file specifies the addresses of all the core-resident programs and a third file contains the addresses for paged programs. The output program accesses these latter two files for address-to-name conversion.

Overwrites account for a major portion of utility system work. An off-line incremental assembler accepts changes in the source language and produces the necessary overwrite data. This procedure greatly reduces the number of errors that might otherwise result. The next step requires that the utility system input the overwrites, allocate space temporarily, and install them for testing. If the overwrites are to be permanent, the utility system allocates the permanent patch space, updates the loader map files, and adds the overwrites to the loader tape.

The output program gathers various types of statistical information about individual test runs. Periodically, through operator request, the

statistical files are output. The statistics include the total number of runs, the number of successful (nonaborted) runs, the accumulated execution time, the accumulated number of input statements, and the accumulated number of output pages.

The utility system employs a message-handling program, whose purpose is to format all messages, to generate a message catalog, to control entering messages and data in the output file, and to maintain a log. Consistency for the utility system output is achieved, since a single program controls all messages and data formats for the output file.

The operator programs provide the operator with the capability to maintain, monitor, and control the 1A Processor utility system. The main operator program, activated by a manual switch, activates any of the other operator programs on request. The operator programs are grouped into administrative and operational programs.

The standard administrative operator programs may be used to initialize the utility computer from the utility system disk, save the disks on tape, reinitialize the disks from tape, update specified files on disk from tape, copy tape to tape, set time and date, and control diagnostic programs for the utility system main frame and peripherals. Special administrative operator programs build or print the statistical files, copy the loader map files from tape to disk, update the loader map files, log permanent overwrites, log the loader tape, and control diagnostic programs for the UTC.

The operational operator programs may be used to activate the input processor program, to terminate the active job, and to cancel any utility system job in progress. A widely used operator program serves as a partial loader. This program temporarily loads any new version of a 1A Processor program from the assembled output module. Other operator programs can dump 1A Processor memory locations or compare the 1A Processor memory with the loader tape. Lastly, since the 1A Processor resident utility program is relocatable, a parameter table defines the addresses of the relocated symbols. This paramater table also contains the operational status of all hardware matchers. An operator program can print or change the value of any parameter in this table.

3.7 Commentary

A convenient and efficient debugging tool has been provided by the 1A Processor Utility System. The batch mode of operation has resulted in a throughput of about 20 jobs per hour with jobs varying in duration from seconds to several minutes. A functional test language has provided the user with a convenient language by which to describe and control tests. Symbolic representation of loader process names has been made available to both input and output. Autonomous tracing and snap-

shotting has satisfied the noninterfering requirement. The overwrite process has been widely and successfully used.

In regard to deficiencies, the memory limitation of the minicomputer has hampered the design, implementation, and throughput of the utility system. The input processor requires three separate tasks to complete its function. Overlapping execution of the input processor, run administration, and output programs to increase throughput can not be realized because of the same memory limitations. Greater use of symbolic representation would be highly desirable; but to accomplish this, access to the common data pool (known as COMPOOL) would be required. Because of the size of COMPOOL (about 100,000 symbols), implementing a capability to reference COMPOOL is not practical in the minicomputer.

To overcome most of these deficiencies, implementation is underway to link the computation center facilities to the minicomputer by means of a data link. As a consequence, the input processor and output programs would execute on the full-scale computer facilities that are available. Only run administration would remain in the minicomputer. It would get the input worklist via the data link from the input processor and in a similar fashion would pass the output file to the output program. This linking to full-scale computation facilities will give the user access to COMPOOL, will allow application of the full power of a high-level language, will allow the loader to install permanent overwrites using source language statements, and finally will improve the test throughput of the ESS laboratories by a factor of two to three.

IV. AN AUTOMATED MODEL-REFERENCED TESTING TOOL

4.1 Model-referenced testing

The first step in developing systematic testing is to determine the standards against which the test is being verified or the behavior model of the process that is being realized by the software. A model at a level more detailed than the rather broad system requirements and guidelines is required. Software testing should not only detect deviations from the modeled behavior but also pinpoint the component causing such deviation.

A fairly accurate and formal class of models is frequently available for software processes, since coding is seldom done directly from the broad system requirements. The operation of the software system in processing a set of external stimuli is typically modeled by a series of small steps. Each step in the model consists of a decision concerning the classification of some external stimuli and/or internal variables followed by a reaction of the software process to the stimuli, external or internal. This model can be easily formalized into a directed graph where nodes represent the decision or stimulus classification points and directed arcs,

representing the outcome of such decisions and the ensuing reaction of the process, join the nodes. A flowchart is a familiar example of such a model; the state diagrams used for No. 4 ESS call-processing functions constitute another set of examples. Further information may be imparted to the model by specifying logical constraints imposed on the process by the physics of the problem. For example, if two classes of stimuli are mutually exclusive in the physical world then the model should exhibit no decisions that classify them into one set. This information is not readily expressed by the adjacency relationships in a graph and needs to be supplied additionally.

The Automated Testing and Load Analysis System (ATLAS) was developed in response to the need to systematically test the complex software processes of the No. 4 ESS. ATLAS provides a means for testing software processes that may be modeled formally by such a directed graph with optional logical constraints imposed upon the order and sequence of arc traversals. Entry (exit) nodes in the graph model usually correspond to the software process getting (relinquishing) control or being initiated (terminated). Every path in the graph that starts at an entry node, terminates at an exit node, and is admissible according to the logical constraints on arc traversals, is composed of a sequence of arcs that may be interpreted as a sequence of steps. Each step describes a stimulus-decision-action-transition sequence. This interpretation of the path description states that if the sequence of stimuli contained in the path is applied to the process modeled by the graph, then the accompanying set of actions must take place. If the stimulus-action descriptions are written as directives to a test driver for generating the prescribed stimuli in the sequence enumerated and for verifying the resultant actions, a test for the software is generated from the model.

The two major functions of ATLAS are to identify the admissible test sequences from the model and to apply the identified tests to the software. These functions and their ancillaries are described in the next two sections. No. 4 ESS application experience of ATLAS as well as the merits and demerits of ATLAS are also given. The interconnection of the components of ATLAS is shown in Fig. 3.

4.2 Test Identification

A directed graph model of the software to be tested is the starting point for the ATLAS process. Some logical propositions are added to the description of the model to express the inherent logic of the physical process being realized by the software. Addition of test commands which cause test drivers to initiate and monitor test actions associated with each arc of the graph makes the model ready for processing by ATLAS.

The first active step in ATLAS processes the model for the identification of admissible tests that are not only consistent with the adjacency

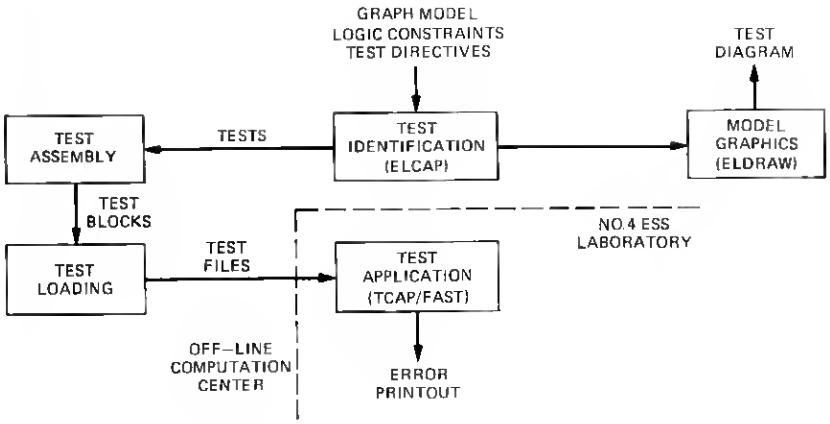

Fig. 3—Components of ATLAS.

of the graph but also satisfy the model logic. This is equivalent to finding paths in the graph that begin at the entry nodes and terminate at the exit nodes and satisfy the logical constraints of the model. The set of all such paths is an exhaustive set of tests for the software as stipulated by the model. This set may be quite large in size for models of even moderate complexity. A smaller set of tests, called the cover set of tests, may be selected from the paths identified. The cover set has the property that every arc of the model is traversed in at least one path of the set. This smaller set of tests is useful for checking the software under test against gross deviations from the model.

The Enumerator of Loops, Covers, and Admissible Paths (ELCAP) is the component of ATLAS charged with the identification of tests from the model. After a thorough syntactic check of the encoded model, ELCAP uses graph-theoretic algorithms to identify the set of all admissible paths in the model and subsequently selects a suitable cover set. Since the presence of loops in the model implies potentially infinite test sequences, ELCAP also identifies all such loops and warns the user about their presence. The traversal of such loops in any admissible path may then be constrained to be finite. The test commands associated with the arcs of each path are collected sequentially and provide a complete specification for the test identified by the path.

The ATLAS component that may be used to draw the encoded model is a computerized graphic process called ELDRAW.

4.3 Test application

The tests derived from the model may be executed in many modes on the actual or simulated software depending upon the test facilities available. These facilities may be the 1A Processor, simulators, or test

drivers constructed specifically for the software under test. The Test Call Application Program (TCAP), a component of ATLAS, is one of the first such test drivers constructed for testing No. 4 ESS call-processing programs. Another component called Facility for the Application of Software Tests (FAST) extends the principles of TCAP to non-call-processing software of No. 4 ESS. Both are ESS resident programs that use the facilities of No. 4 ESS to apply and to monitor tests for the software in as natural a manner as possible.

TCAP has the capability of emulating various sources of stimuli that are recognized by the No. 4 ESS call-processing system. Software mechanisms enable TCAP to secure control before and after these stimuli are acted upon by the software under test so that TCAP may verify the accuracy of these actions against those specified in the test. Abnormal conditions, such as resource blocking, are also simulated by TCAP.

Commands associated with a test are presented to TCAP as a data block, called a test call block. Entries in this block generally contain command codes followed by command operands and are interpreted by the routines of TCAP. The execution of a test via TCAP may be described as follows. The initialization data provided in the test call block enables TCAP to secure the required trunks for the test to be executed. TCAP assumes control and generates stimuli on the remote ends of these trunks. Other necessary initialization is also done. Testing is then started by generating the first stimulus, generally a call origination prescribed in the test call block. When the stimulus is reported to the operating system of ESS, a built-in software mechanism transfers control to TCAP. TCAP checks the accuracy of the report, sets up linkages so that control may be returned to TCAP after the software under test has had a chance to process the stimulus, and then invokes the call-processing software. When control is regained by TCAP, verification commands in the test call block direct TCAP to monitor the accurate processing of the stimulus. The cycle is repeated for each stimulus until a test termination command is reached or the software is detected to behave differently from the pattern given in the test call block. The test is then terminated with the appropriate success or failure message. The system is subsequently purged of the test.

TCAP is capable of generating a large number of tests in parallel. Various other capabilities, such as timing and manipulation of special call-processing data and control structures, are also provided.

The Facility for the Application of Software Tests (FAST) extends the basic pattern set by TCAP to other No. 4 ESS software. The requirements are the same in both cases; generation of stimulus in a natural fashion, emulation of abnormal conditions, verification of response, and a smooth control interface between the operating system, the test driver, and the programs under test. Due to the wider scope of FAST, the control inter-

faces are not built into the system. They are generally realized via 1A Processor Utility System commands. FAST provides some general functions, e.g., stimuli generation, data initialization, timing, simulation of some systemwide abnormal conditions, and general data verification. It also provides for linkages to any special-purpose routines the user may generate so that the capabilities described above may be customized for the application at hand.

Two specialized command languages, one for each of the two test drivers described above, are incorporated in ATLAS to give the user the capability of writing test commands in a symbolic and easily readable format. The constructs in the languages are chosen to be particularly relevant to the application. The languages have declarative statements that permit the associated test drivers to initialize tests as well as executable statements that actually direct test actions. Since tests written in these languages consist of strictly predetermined sequences of stimulus-action pairs, conditional statement facilities are not provided. The conditional aspects of the tests are handled in the identification stages by ELCAP.

Assemblers, constructed with the macro handling facilities of SWAP, the assembler for ESS programs, are provided in ATLAS for resolution of symbols. These assemblers use the same symbol dictionaries (COMPOOLS) that are employed by the program assemblers so that ATLAS users may maintain uniform symbol definitions in both the coding and testing phases. The assemblers allow for initialization declarations to be placed in the model where deemed relevant by the user. They also bind tests to models via test and arc identifiers to facilitate the administration of tests and to resolve errors identified by the tests. A loader handles the movement of high volumes of test data generated in the computation center facilities (where parts of ATLAS are resident) to the No. 4 ESS Laboratories where TCAP and FAST are resident.

4.4 Usage data and summary

The components of ATLAS have been used in various combinations to test over 40,000 instructions in No. 4 ESS software. In the majority of cases, ATLAS was used as a testing tool during the phase of integrating various component programs of a software function. The call-processing system has used ATLAS extensively. In all cases, the programs had been tested by using nonautomated methods before ATLAS testing methods were employed, though only a small set of cover tests were run rather than an exhaustive set of tests. The incompleteness and inconsistencies of the models constituted a large class of errors. However, about 300 problems were found in the pretested software during ATLAS testing, excluding the model errors. The errors ranged from hardware problems

that surfaced because of the intense mode of testing by the test drivers to software logic and implementation errors.

The primary benefit derived from ATLAS is the formalization of the testing process. Programmers are relieved from the very detailed, tiresome, and error-prone work of specifying how to test their software so that more thought may be given to the fundamental question of what needs to be tested. The repeatability of the tests, testing in a natural environment, documentation, and ease of test administration are other benefits that are directly attributable to the formalism imposed by ATLAS.

Since ATLAS is a model-referenced testing method, the excellence achievable by ATLAS testing is inherently dependent upon the quality of the model. If the model is incomplete in reflecting the requirements of the software, the set of tests derived from it will also be incomplete. Furthermore, since ATLAS cannot distinguish between a model error and a genuine program bug, model inconsistencies will generate spurious error data. These facts imply that the model should be carefully developed and checked for completeness and consistency for ATLAS testing to be fully effective. The use of a formal approach always needs more thoughtful planning than its ad hoc counterparts. Lead time to do this thinking and planning is needed. The direct computer charges associated with the use of an algorithmic testing approach such as ATLAS can be expected to be high. The software product of the No. 4 ESS project is designed to have high reliability and is expected to undergo considerable change over the years as new capabilities are added to the system. The benefits of formalism and discipline inherent in the use of ATLAS far outweigh its disadvantages.

V. PROGRAMMED ELECTRONIC TRAFFIC SIMULATOR

5.1 System description and environment

The stress testing and evaluation of large real-time software systems present unique problems. System performance under load is difficult to measure and is difficult to simulate. Stress testing of an Electronic Switching System (ESS) is no exception.

Previous ESS traffic simulation methods have employed electromechanical load boxes and software simulators with simulated calls applied to the network terminals. This approach required large quantities of per-terminal equipment and a network with associated peripherals large enough to process the high traffic loads. For large systems with high call capacities, terminal simulation processors cannot be economically justified because of both the dollar expenditure for all of the required equipment and the space limitations of laboratory test models. Thus, in order to apply a realistic traffic load to a No. 4 ESS in a laboratory

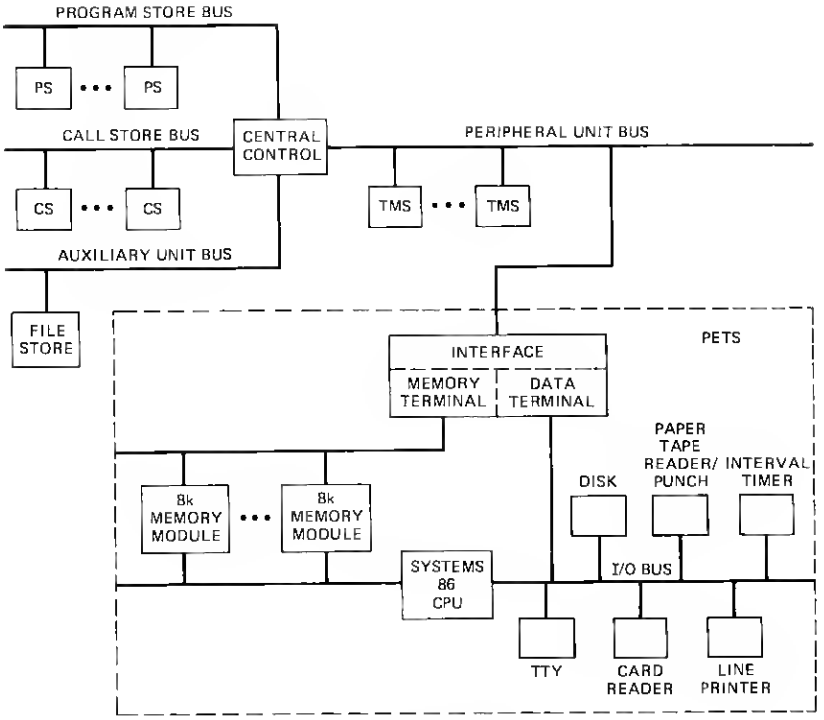

Fig. 4—PETS system.

environment, a new approach was necessary. The search for a new approach led to the development of PETS—the Programmed Electronic Traffic Simulator.

The basic objectives of the simulator system are the following:

(*i*) Generation of traffic levels that exceed the designed or engineered capacity of the system.

(*ii*) Interconnection with ESS via the Peripheral Unit (PU) bus.

(*iii*) Generation of random traffic loads that are realistically distributed in time.

(*iv*) Generation of traffic that is independent of the available ESS peripheral hardware.

(*v*) Operation in real time.

(*vi*) Operation without requiring any significant modification to the ESS program.

(*vii*) Collection of sufficient meaningful data to allow evaluation of ESS performance.

To meet these objectives, the simulation system illustrated in Fig. 4 was developed. The PETS system consists of a commercial computer, a hardware interface which connects the simulator to the peripheral bus

system of the No. 4 ESS, and a memory and data terminal which enable the interface to communicate with the commercial computer. These components are discussed in more detail in subsequent sections of this paper.

PETS simulates most of the environment external to the ESS. To accomplish this, the peripheral equipment action that would result from either connecting office stimuli or the peripheral equipment action that would result from the ESS response must be generated. The outputs from the ESS processor to the peripheral system must be intercepted and analyzed. These outputs are compared with expected responses to analyze the performance of ESS.

PETS does not replace all of the peripheral equipment in a laboratory but supplements the existing laboratory peripheral system by making it appear to ESS that it is working with a large office.

To meet the stated objectives, the system described in the following paragraphs was developed. The type of peripheral equipment simulated includes the Signal Processor (SP), the Time-Slot Interchange (TSI), and the Terminal Access Circuit (TAC). Because the system laboratories are equipped with the necessary Time-Multiplexed Switch (TMS) configurations, this portion of the network does not require simulation.

To simulate the proper environment, both the PETS and the No. 4 ESS data bases must identify the peripheral equipment of a large office and must contain corresponding translation and parameter data. The real peripheral equipment in the laboratory may be defined in the ESS data base along with the simulated equipment. This allows real and simulated traffic to be simultaneously presented to the ESS processor.

The amount of memory required for the PETS system depends, in part, on the number of simulated circuits. Each simulated terminal, whether it is a trunk or service circuit, requires a dedicated block of storage in the PETS memory. To keep the memory required for PETS manageable, a relatively small number of terminals is simulated instead of a full-size network. In addition, the simulated terminals are evenly spread over the simulated equipment.

To achieve the large traffic volume necessary to adequately test the ESS, the per-terminal origination rate must be higher than the per-terminal origination rate of a working office. Thus, to provide a high calling rate using a reduced number of terminals, shorter holding times or "talking" intervals are necessary. However, realistic distributions of other call timing intervals are maintained.

PETS has the capability of simulating a maximum of eight SPs having both universal and miscellaneous scan and signal distributor points and eight additional SPs having only miscellaneous points. The additional miscellaneous points are required for the MF transmitters and receivers for maximum MF traffic loads.

The PETS hardware simulates a maximum of five TSI frames and four TACs with 16 terminals each. Only the TAC low-priority and high-priority message buffers are simulated.

Communication between the No. 4 ESS processor and the PETS system is via the peripheral unit bus system. This system consists of the enable address bus, the write bus, the reply bus, and tbe control bus. In addition, there is a communication link between the utility test console and the PETS system for start/stop control in either direction.

A hardware interface unit provides for the connection to the ESS peripheral unit bus system and the transfer of data into and out of simulator memory.

The computer used in the PETS system is the SEL Systems 86. In addition to the normal peripheral configuration, the Systems 86 is equipped with a special memory terminal and data terminal. The memory terminal is utilized by the interface to gain access to the Systems 86 core memory. The data terminal provides for the transfer of status information from the interface to the PETS software system.

To provide for the transfer of data between the interface and the PETS programs, two-port memory is utilized. This allows the interface to retrieve data directly from dedicated areas of memory and to store data necessary for processing by the software. The programs, in turn, can update status information utilized by the hardware interface. The area of core accessed by both the interface and the programs is referred to as "shared memory."

Control and status information is exchanged between the interface and the programs via the data terminal. The control functions of starting and stopping the systems are provided through the use of the data terminal.

PETS must provide external stimuli to ESS such as originations, dialed digits, answer, and disconnect signals. To accomplish this, the simulation program enters data into the SP buffers and updates simulated T-bits and scan points in shared memory. When ESS transmits an order to read a buffer, T-bit data, or scan points, the interface intercepts the order, retrieves the data from the proper locations of shared memory, and returns the data to ESS.

In response to the stimuli, the ESS processor sends orders to the peripheral equipment to connect and disconnect network paths, outpulse digits, etc. The interface intercepts these orders, provides the proper response to ESS and stores appropriate data in a common buffer area of shared memory. The simulation program periodically processes the data in the common buffer. The simulated portion of any equipment affected by the order is updated in the proper time sequence and appropriate additional stimuli are provided to ensure the progress of the simulated call.

The simulation program provides a random distribution of originations that approximate those encountered in a working office. In addition, realistic timing variations are provided for the various interoffice signaling stimuli and responses.

Call characteristics, such as permanent signal, partial dial, call abandon, and originator on-hook, are provided on a random basis based upon the parameters specified for a simulation run.

The simulation program keeps records of the traffic presented to ESS and the disposition of that traffic. Records are kept of all originations, terminations, anomalies and unexpected events. In addition, histogram data is maintained for the time delays between the various stages of call setup. The data are used to evaluate the performance of the ESS under load.

The number of terminals in the simulated office is dependent upon the desired load, the average call-holding time, the amount of available ESS call store, the amount of available PETS memory, and the number of service circuits required for the traffic load. The ESS call store requirement is dependent upon the maximum load and the number of simulated terminals. Based, in part, on these considerations, the system laboratory is engineered and the Office Data Assembler (ODA) is utilized to provide the ESS translation and parameter data required for the simulated peripheral equipment.

A data assembler produces the translation data base needed by the PETS system. The input to the data assembler will be derived from the same input used by the ESS ODA.

The translation and parameter data must be supplemented with additional data describing the call types to be included in the simulation run, the desired load characteristics, the various call anomalies to be simulated, etc. Much of the data is processed and built off-line. Thus, very little run-time processing is necessary.

5.2 Hardware configuration

The hardware interface provides the communications path between ESS and the PETS system. The functions of the interface are to:

(*i*) Recognize when ESS sends a peripheral order to the simulated equipment, extract the appropriate data, and gate the data to the software system.

(*ii*) Process, via hardware, those orders requiring an immediate response to the ESS.

(*iii*) Provide the necessary timing and control functions to properly transfer or exchange data between the ESS PU bus and the Systems 86 Central Processing Unit (CPU).

(*iv*) Provide logic level shifting and data formatting for efficient

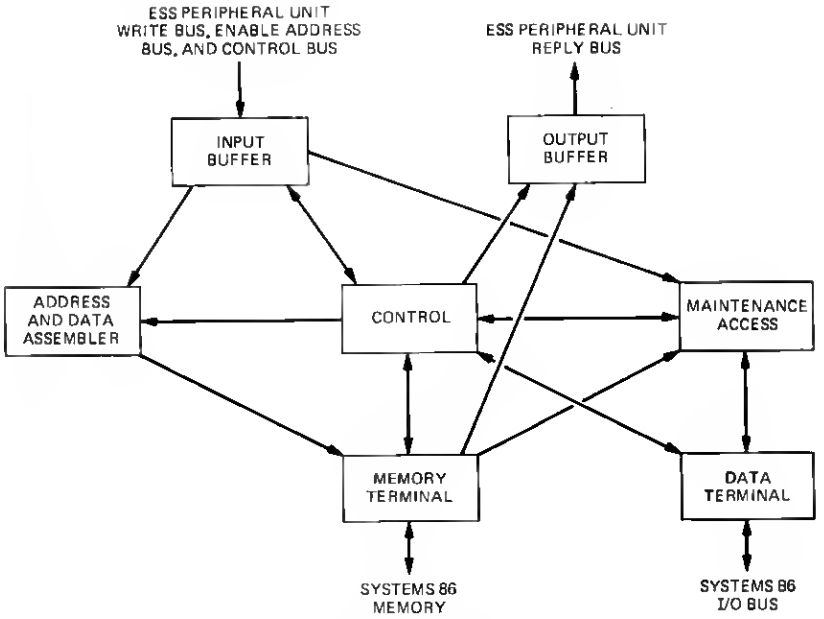

Fig. 5—PETS interface.

transfer of information between the ESS PU bus and the Systems 86 CPU.

(*v*) Generate, upon demand, signals to control both the ESS processor and the PETS system clock.

(*vi*) Provide a visual display of the interface status and report all anomalies in operation to the software system.

A block diagram of the interface is shown in Fig. 5.

The PETS interface with ESS is at the PU bus. Modified ESS cable drivers and receivers are employed. The modifications provide the necessary interface between the +1-volt logic levels of the ESS circuits and the +5-volt logic levels of the TTL circuits employed in the interface.

The interface processes those peripheral orders accompanied by the address of a simulated peripheral frame. When the address match is detected, the information on the PU write bus and enable address bus is processed. The interface also responds to the pulses on the PU control bus. These pulses are stored in the control register until the interface has appropriately responded to the ESS processor.

Responses from PETS to ESS are transmitted over the PU reply bus. The interface generates a response which is identical to replies from the real peripheral frames. An all-seems-well pulse and a parity bit are also transmitted as part of each response.

The exchange of information between the interface and the Systems 86 CPU is accomplished via two individual channels of communication. The memory terminal enables the interface to access any part of the Systems 86 memory and provides the capability to read or write as many as 64 bits (double-word) of data. In addition to double-word transfers, the memory terminal provides the capability of word, halfword, byte, and bit transfers.

The interface must provide the memory terminal with an address compatible with the Systems 86 memory address format. This address is derived from three sources:

(*i*) Information received by the interface from the peripheral unit bus.
(*ii*) Hardware address counters in the interface.
(*iii*) Information obtained from a previous memory-read operation.

Approximately 24K words of Systems 86 memory are utilized as shared memory. The interface accesses this memory to obtain the data associated with the simulated equipment and to obtain address pointers and indices necessary to locate the data in shared memory.

Since PETS is designed as a load simulator and not as a maintenance tool, no faults or other hardware anomalies are simulated. Thus, the interface simulates responses to most of the operational orders and a small subset of maintenance orders associated with the simulated equipment.

The simulation of peripheral orders may require as many as five separate accesses of shared memory. For example, on a "read SP buffer" order, the interface must first read the dedicated location associated with the particular SP and buffer type to obtain the pointer to the next buffer entry to be unloaded. These data are then used to read the particular buffer entry into a register in the interface. The interface must then zero the memory location where the buffer entry was previously stored. Next, the interface must update the address pointer and write the new pointer into the dedicated address. Finally, the buffer entry is gated onto the reply bus for transmission to ESS.

The PETS interface operates asynchronously from both the ESS CC and the Systems 86 CPU. The timing and control functions are derived in the interface from data provided by the ESS central control and the Systems 86 CPU. The basic clock signal used by the interface is derived from a clock signal transmitted from the Systems 86 CPU via the memory terminal.

The interface detects certain types of failures during processing of peripheral orders and generates an interrupt through the data terminal

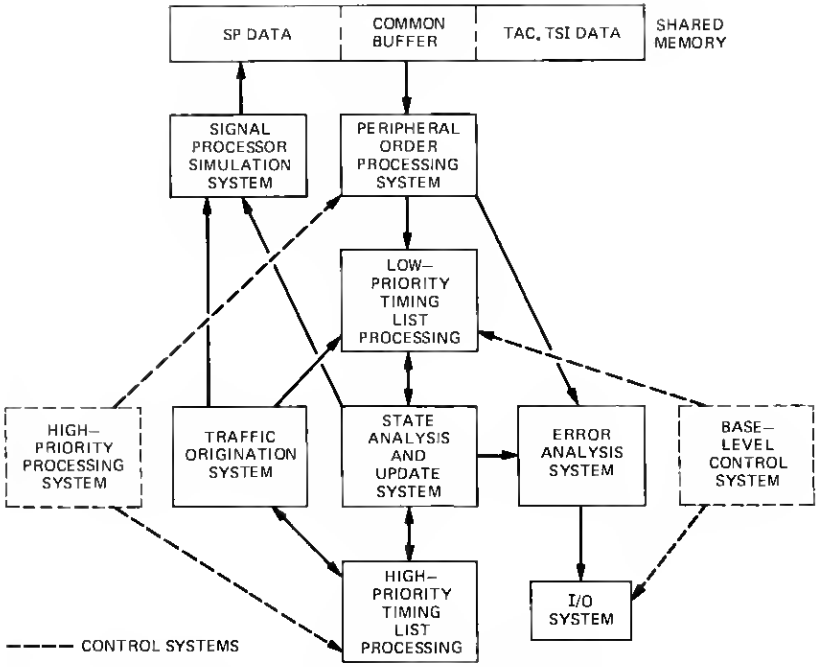

Fig. 6—Functional diagram of PETS software.

to notify the software system of the problem. The types of failures detected by the interface are:

(*i*) An attempt to address nonpresent memory.

(*ii*) A parity error occurring during a read operation of shared memory.

(*iii*) An attempt to address a protected area of memory.

The cause of the failure is determined by the software utilizing test instructions.

5.3 Software description

The PETS software consists of many systems—each performing a generalized function. Figure 6 is a functional diagram of the software illustrating the various systems, with the dashed lines indicating the control programs. The following paragraphs describe the systems shown in the figure.

Data associated with the receipt of ESS peripheral orders are stored by the interface in a common buffer in shared memory. The peripheral order processing system periodically unloads this common buffer and processes each entry. Most orders are processed immediately. However,

in the case of network orders, processing is deferred until a complete ESS sequence (e.g., connecting an originating MF trunk to a receiver) has been received.

The signal processor simulation system administers the four buffers associated with each SP. This system provides the loading of these buffers with the various reports required during the simulation of a call and provides the ability to temporarily store a report if the buffer does not presently contain an empty slot.

The PETS software system is organized around two levels of processing—high-priority and base-level. The high-priority processing system utilizes a hardware timer that provides an interrupt every 5 ms. This interrupt causes the system to be entered for unloading the common buffer and the processing of peripheral orders. All high-priority tasks requiring attention are also processed during this interval. At the conclusion of high-priority processing, control is returned to base-level processing at the point of interrupt.

The base-level control system provides the basic control for the continuous cycling of the base-level tasks including base-level timing list processing and I/O control. To ensure that PETS is providing proper timing resolution, the length of the base-level cycle is continually monitored to ensure that it does not exceed specified limits.

The timing list processing system provides the real-time breaks between segments of a call simulation and provides the ability to measure various events occurring during a simulation run. Use of both high-priority and base-level timing allows great flexibility in the amount of time requested and the precision assigned to each request.

The state analysis and update system analyzes the various ESS responses or lack of response, determines the next state of the call simulation process, and provides ESS with any necessary stimuli.

The traffic origination system determines the time interval between call originations and generates Poisson traffic based upon data supplied by the user. This system determines the originating trunk and makes various call simulation decisions (e.g., abandon, permanent signal, etc.) associated with the call and presents the stimuli to ESS to accomplish the origination.

All error messages, status information, and user communications are processed by the I/O system. This system has the ability to process input commands and to supply the necessary data and communications without stopping or interfering with the simulation run.

All unexpected ESS responses are processed by the error analysis system. Data is recorded for output and all necessary action is taken to recover from the situation.

The philosophy for the software design is based upon three major concepts: the call register, timing list processing, and state analysis.

Keeping in mind the previously described systems, a general description of the PETS software follows, beginning with the call register.

Every simulated call in the system has a 6-word block of memory associated with it for the duration of the call. The call register contains the up-to-date status of the simulated call including the terminals involved, decision information to guide the progress of the call, and an indication of all ESS actions or responses received since the call register was last processed. Call registers are seized from an idle list during the origination and released after the call has been completely processed. During actual simulation, the call register will be linked to a timing list waiting for appropriate ESS actions or for a time-out to occur before presenting the next stimuli to ESS.

Each call register has an associated state stored as an item in the call register. The state value determines which program will process the call register after either the last requested time delay has elapsed or when ESS has taken some sort of action that involves the call. If an ESS action is involved, the processing of the call register will take place during base level; if a time-out occurs, the processing will take place in either high-priority or base level depending upon the timing precision required.

From an examination of the data contained in the call register, it can be determined what terminals are involved, how far the call has progressed, what ESS action has been received, what state analysis program to use to process the call register, and what anomalies, if any, are to be simulated as part of this call.

Numerous real-time breaks and timing measurements are required for each simulated call. Two timing lists are utilized. Base-level timing lists provide a wide range of timing applicable for most situations; high-priority timing lists provide greater precision for those events requiring critical timing but at the expense of being processed during the high-priority interrupt cycle.

Each timing list processing system consists of four timing list tables. The first table of the high-priority timing list is examined every 5 ms during high-priority processing. That is, every 5 ms the next slot of the first table is examined and all call registers linked to that slot are analyzed and processed.

Every time the processing of the first table is recycled (i.e., slot 0 is processed), the next slot of the second table is processed. Similarly, when the second and third tables are recycled, the next slot of the third and fourth tables, respectively, are processed. Thus, with four tables with 16 slots each, up to 5.461 minutes of high-priority timing in 5 ms increments can be provided.

Base-level timing differs from high-priority timing in that the processing is done during base level, the increment of time between slots on the first list is 20 ms instead of 5 ms, and the first list is 64 slots in

length instead of 16. The base-level timing list tables provide up to 1.456 hours of timing in 20-ms increments.

State analysis is a set of programs that analyze a call register to determine how far the call has progressed and what action is necessary next. Each call register has an associated state, and it is this state value that determines which program to use to process the call register. A state analysis program is entered as a result of processing or examining a call register and determines what stimulus, if any, is required for ESS or what time delay is necessary before taking any further action.

Functions other than those associated with the processing of a call can be provided by using special registers with states corresponding to the appropriate function.

Traffic originations are the result of processing special registers. When one of these traffic origination registers is processed, a traffic origination program is entered to generate a call and to make a decision, based upon the specified calling rate, as to when the next origination should occur.

Each of the traffic origination registers contains a calling rate that is currently in effect. The processing of another set of special registers causes the calling rate stored in the traffic origination registers to be periodically updated. Thus, a given calling rate curve can be approximated by a table containing 256 equal time intervals, each of which has a constant calling rate corresponding to the average over the interval. At the completion of each time interval, the next calling rate is retrieved from the table and stored in the traffic origination register. All traffic originations, selection of call characteristics, and trunk selections are based upon random number generators. Thus, the PETS system approximates user-specified values for the various call characteristics on a "random" basis.

A monitor system provides user communications prior to the actual start of the simulation run, provides the initial linkage of all call registers to the idle list, controls the PETS-ESS start/stop communications, processes all interrupts resulting from abnormal conditions (e.g., parity failure, nonpresent memory, etc.), and controls the periodic (5 ms) interrupt. The monitor system also controls the starting of the simulation run and the duration of the run.

5.4 Data accumulation

During a simulation run, the system records various events for analysis of ESS performance. The type of data collected is described in the following paragraphs.

Originations are recorded on an originating-trunk-type basis and on a trunk-subgroup basis. An origination is recorded when the initial SP buffer entry is made. Terminations are recorded at both the path res-

ervation stage and at the path setup stage of a call and are recorded according to the type of call and on a trunk-subgroup basis.

Counters are incremented as the simulated calls progress through the various stages and paths of call processing. In addition to recording completed calls, counters are incremented as each call anomaly (e.g., abandon and permanent signal) is simulated.

In order to evaluate the performance of ESS, PETS records those events related to the resources available to ESS. Counts are kept for all connections to tones and announcements, for events that indicate no available MF receiver or ESS call register, for failure to connect to a tone or announcement when one is required, for overflow of SP simulated buffers, and for similar items reflecting on ESS performance and the engineering of the ESS system.

Evaluating the performance of ESS requires measurements such as incoming delay, seizure time, address time, response time, answer time, and disconnect time. These six critical timing measurements are vital indicators of ESS performance. In order to record useful data and to display the data in meaningful form, these critical timing measurements are accumulated in histogram form. Each histogram consists of 16 elements of equal interval.

The PETS software system is in itself a complex real-time operating system with data that must be engineered to correspond to the simulated office environment. The system monitors and records its own base-level and high-priority cycle times, the number of times no trunk or no call register was available for an origination and the number of times the system was not able to cause ESS to properly idle a trunk.

PETS provides a periodic traffic summary in matrix form indicating the number of originations on a trunk-type basis and the disposition of those originations (e.g., completions on a trunk-type basis, abandons, and connections to tones and announcements). This traffic summary is output every minute unless the user specifies a different interval, and each summary presents the data for the last time period.

Because of the large volume of data collected during a simulation run, it is neither desirable nor practical to output a hard copy of this data during the run. The PETS system provides the ability to periodically dump the data to a disk file for analysis at the conclusion of a simulation run. A comparative analysis is available to print the data associated with each period and to compare one period with another. The data are presented in a well formatted dump indicating the counter description, its internal mnemonic, and the associated value.

5.5 Application results

Providing for the testing and evaluation of ESS performance is a major goal of the PETS system. In addition, the simulation system is designed

to be used as a tool for supplying background loads for the debugging, integrating, and testing of ESS programs and systems. Some of the many applications and results are discussed in the following paragraphs.

In June, 1975, PETS was used to experimentally verify that No. 4 ESS could process its design objective peak busy-hour call load of 385,000 switched calls (431,000 total call attempts) while meeting all speed-of-service criteria. During the experiment, all normally scheduled maintenance tasks were performed as well as traffic administration and network management functions. Service circuits and transient call memory were engineered for the expected load. The results of this experiment demonstrated overwhelmingly that No. 4 ESS met and generally exceeded the advertised busy-hour capacity.

As a result of the initial experiment, further experiments were made that verified a new advertised peak capacity of No. 4 ESS of 550,000 switched calls (616,000 total call attempts) per busy-hour.

In addition to providing capacity verification results, PETS is a valuable tool for providing background loads for system testing. It has provided background loads exceeding 630,000 call attempts per hour while testing and evaluating the recent change system, the network management system, the traffic measurement system, and trunk maintenance procedures.

With background loads, more exhaustive testing of No. 4 ESS was possible, and means were available to compare internal ESS data with corresponding data in the PETS system.

PETS has been used to provide various loads to No. 4 ESS while measuring different aspects of performance. Many performance measurements such as base-level cycle times were taken and many logic and programming errors were uncovered during this load testing.

The present real-time capacity of the simulator is in the range of 680,000 call attempts per hour. It is quite possible that this capacity could be significantly increased by eliminating much of the internal data collection and by streamlining some of the more frequently used programs.

The PETS concept of load simulation has been successfully applied to the development of No. 4 ESS. The same concept with its associated rewards can be applied to other real-time software systems.

VI. CONCLUSION

Large systems such as No. 4 ESS, which evolve as new capabilities are added over a long period of years, must continue to have support systems like those described in this paper. In fact, the support systems themselves must continue to evolve in order to meet the ever-increasing challenge arising from the need to produce a high-quality product under more and more difficult circumstances. Difficulties arise from the need

to support multiple generics which are inherent to the evolution process, the attendant growth in size, and the need to provide support to the field.

As we look to the future in No. 4 ESS support systems development, we see a three-pronged approach: (*i*) The scope of support must be widened to include all aspects of software development from the determination of requirements to the administration of code in the field. (*ii*) The individual support systems must be integrated into a network of systems which allows each process access to the necessary data of the other processes and which provides the user with a single, uniform interface. (*iii*) Such a network of systems must facilitate the introduction of new processes and allow the modification of those that already exist.

In summary, the four support systems described herein have significantly assisted the development process which produced the initial versions of No. 4 ESS. Together with No. 4 ESS, these support systems themselves will continue to evolve in order to provide new capabilities and to meet the demand for greater development efficiency.